# DLT Project Report

### Model comparison for LSTM and Feedfoward with Amazoen review text data

성균관대 통계학과 박나린 (석사) 2022711906

---

# 0. Setting

케라스 기반의 딥러닝 모델들을 쓰기 위한 기본 환경 설정을 해준다. 또한 파이썬 기반의 모형들을 쓰기 때문에, 파이썬에서 처리를 위한 패키지들도 불러 와준다. 이 프로젝트에 사용된 데이터는 캐글에서 구한 Amazon Reviews for Sentiment Analysis 이다.

출처 :: https://www.kaggle.com/datasets/bittlingmayer/amazonreviews

코딩과 함수들은 수업 자료들과 다양한 코드들을 참고 하였으며 특히 6장 sequential model에 관한 데이터 부분을 참고하여 쓰여진 보고서 이다.

In [1]:
```
import pandas as pd
import numpy as np
import bz2
import matplotlib.pyplot as plt
import sklearn

import tensorflow as tf
from tensorflow.keras import models, layers, optimizers
from tensorflow.keras.preprocessing.text import Tokenizer, text_to_word_sequence
from tensorflow.keras.preprocessing.sequence import pad_sequences

%matplotlib inline
```

In [6]:
```
import os
for dirname, _, filenames in os.walk('C:/Users/USER/Dropbox/data/input2'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
C:/Users/USER/Dropbox/data/input2Wtest.ft.txt.bz2
C:/Users/USER/Dropbox/data/input2Wtrain.ft.txt.bz2
```

In [46]:
```
tf.__version__
```

Out[46]:
```
'2.12.0'
```

In [ ]:
```

```

# 1. Data Processing

이 데이터셋은 fastText를 학습하기 위한 아마존 고객 리뷰 몇 백만 개(입력 텍스트)와 별점(출력 레이블)으로 구성되어 있다. 이 데이터셋은 양질의 비즈니스 데이터로 실제 규모이지만, 소규모 노트북에서 몇 분 안에 학습할 수 있도록 구성되었다. train 과 test 파일이 각각 존재 하므로 train test를 나눌 필요가 없다.

---

## 1.1 Import Data

먼저 bz2 파일 형식으로 되어있는 train, test 파일을 불러온다

In [7]:
```
# bz2 데이터 읽어 오는 함수 만들어주기

def labels_texts(file):
    labels = []
    texts = []
    for line in bz2.BZ2File(file):
        x = line.decode("utf-8")
        labels.append(int(x[9]) - 1)
        texts.append(x[10:].strip())
    return np.array(labels), texts
```

In [9]:
```
train_label, train_text = labels_texts('C:/Users/USER/Dropbox/data/input2/train.ft.txt.bz2')
test_label, test_text = labels_texts('C:/Users/USER/Dropbox/data/input2/test.ft.txt.bz2')
```

In [10]:
```
print(train_label[0])
print(train_text[0])
```

```
1
Stuning even for the non-gamer: This sound track was beautiful! It paints the senery in your mind so well I would recomend it ev
en to people who hate vid. game music! I have played the game Chrono Cross but out of all of the games I have ever played it has
the best music! It backs away from crude keyboarding and takes a fresher step with grate guitars and soulful orchestras. It woul
d impress anyone who cares to listen! ^_^
```

In [11]:
```
import re

not_numChar = re.compile(r'[WW]')
no_encode = re.compile(r'[^a-z0-1Ws]')

def normalisation(texts):
    norm_text = []
    for word in texts:
        lower = word.lower()
        not_punct = not_numChar.sub(r' ', lower)
        exclude_no_encode = no_encode.sub(r'', not_punct)
        norm_text.append(exclude_no_encode)
    return norm_text
```

## 1.2 Normalize text data

텍스트를 처리하기 위해 첫 번째로 수행할 작업은 모든 문자를 소문자로 변환하고, 단어가 아닌 문자를 제거 하는 것이다. 이때 대부분의 경우 이는 구두점으로 대체된다. 그런 다음, 강세 기호가 있는 문자와 같은 다른 문자를 제거한다.. 이러한 문자들 중 일부는 정규 ASCII 문자로 대체하는 것이 더 좋을 수도 있지만, 여기서는 그것을 무시하기로 결정했다.

---

In [12]:
```
train_text = normalisation(train_text)
test_text = normalisation(test_text)
```

In [13]:
```
print(train_text[0])
```

```
stuning even for the non gamer  this sound track was beautiful  it paints the senery in your mind so well i would recomend it ev
en to people who hate vid  game music  i have played the game chrono cross but out of all of the games i have ever played it has
the best music  it backs away from crude keyboarding and takes a fresher step with grate guitars and soulful orchestras  it woul
d impress anyone who cares to listen
```

In [14]:
```
y_train = np.array(train_label)
y_test = np.array(test_label)
print(y_train.shape)
print(y_test.shape)
```

```
(3600000,)
(400000,)
```

In [ ]:
```

```

## 1.3 Tokenize the data

토크나이즈(tokenize)는 텍스트를 작은 단위로 분할하는 과정으로 토크나이즈를 통해 텍스트를 단어 단위로 분할하면 자연어 처 리 작업에 대해 더욱 정확한 분석이 가능해진다. 각각의 단어는 개별적인 의미를 가지며, 이를 활용하여 문장 또는 문서의 의미 를 파악하거나 감성 분석과 같은 작업을 수행할 수 있는 구조를 뛰게 해준다.

---

In [ ]:
```
max_features = 8192 # We will only consider the top 8,192 words in the dataset
maxlen = 128  # We will cut reviews after 128 words
embed_size = 64

tokenizer = Tokenizer(num_words=max_features)
tokenizer.fit_on_texts(train_text)
```

In [ ]:
```
training_token = tokenizer.texts_to_sequences(train_text)
testing_token = tokenizer.texts_to_sequences(test_text)
```

In [ ]:
```
val_comments = tokenizer.texts_to_sequences(val_comments)
```

## 1.4 Padding

패딩은 모든 입력 시퀀스가 동일한 길이를 갖도록 하는 작업으로 자연어 처리 작업에서는 텍스트 분류나 시퀀스 생성과 같은 많은 작업에서 기계 학습 모델을 훈련시키기 위해 고정 길이의 입력 시퀀스가 필요핟.

패딩은 짧은 시퀀스에 특수 토큰(보통 0)을 추가하여 데이터셋에서 가장 긴 시퀀스와 길이를 맞춰준다. 이렇게 하면 시퀀스들을 행렬로 쌓아 모델에서 효율적으로 처리할 수 있게하기 때문이다. 패딩을 사용하지 않으면 시퀀스들은 각각 다른 길이를 갖게 되어 대부분의 기계 학습 알고리즘에서는 고정 크기의 입력을 예상하기 때문에 적합하지 않게된다.

이 프로젝트에서는 pad_sequences 함수를 사용하여 훈련과 테스트 시퀀스(training_token 및 testing_token 각각)를 지정된 길이(maxlen)에 맞게 패딩한다. 패딩은 각 시퀀스의 끝에 0을 추가함으로써 이루어지게 되고(padding='post'), 이렇게 하면 모든 입력 시퀀스가 동일한 길이를 갖게 되어 모델에서 처리할 수 있게 만들어준다.

---

In [ ]:
```
x_train = pad_sequences(training_token, maxlen = maxlen, padding = 'post')
x_test = pad_sequences(testing_token, maxlen = maxlen, padding = 'post')
```

In [ ]:
```

```

# 2. Model Fitting

## 2.1 CNN

CNN은 "Convolutional Neural Network"의 약자로, 합성곱 신경망이라고도 불린다. 이는 주로 이미지 및 시계열 데이터와 같은 그리드 형식의 입력에 사용되는 신경망 아키텍처로 특히 이미지 인식, 컴퓨터 비전 및 자연어 처리와 관련된 작업에 많이 사용된다.

CNN은 입력 데이터의 지역적인 패턴 및 구조를 인식하기 위해 컨볼루션 연산을 사용하는데 이는 입력 데이터를 작은 윈도우로 나누고, 각 윈도우에 대해 필터와의 컨볼루션을 계산하는 것을 의미한다. 이러한 컨볼루션 연산은 데이터의 특징을 추출하는 데 도움이 된다. 컨볼루션 연산의 결과는 활성화 함수를 통과한 뒤, 풀링(pooling) 레이어를 통해 다운샘플링 되고, 이 과정을 반복하여 신경망은 점차 더 추상적인 수준의 특징을 학습하게 되는 것이다.

CNN은 이러한 컨볼루션 및 풀링 레이어의 조합으로 구성되며, 일반적으로 여러 개의 컨볼루션 레이어와 풀링 레이어가 번갈아가며 쌓이는 형태를 가지고 있다. 이후에는 전체 특징 맵을 완전 연결(fully connected) 레이어로 연결하여 최종 분류를 수행하게 된다.

CNN은 이미지 처리 작업에서 좋은 성능을 보이며, 최근에는 자연어 처리 작업에도 적용되고 있다. 텍스트 분류, 감성 분석, 문장 생성 등의 자연어 처리 작업에서 CNN은 텍스트의 지역적인 구조와 패턴을 학습하여 효과적인 모델을 구축하는 데 도움이 된다.

따라서 먼저 CNN구조의 신경망을 사용하여 모델 적합을 해보기로 하였다.

---

- **사용한 CNN 모델**

이 모델은 64 차원의 임베딩을 가지고 있다. 3개의 합성곱 레이어가 있으며, 첫 번째와 두 번째 레이어는 배치 정규화(batch normalization)와 최대 풀링(max pooling)을 사용하고, 마지막 레이어는 전역 최대 풀링(global max pooling)을 사용한다. 최종 결과는 밀집 레이어에 전달되었다.

옵티마이저로 RMSPROP을 사용하고, 이는 분류 문제이므로 손실 함수로 이진 크로스엔트로피(Binary Crossentropy)를 사용했다.

train 데이터를 이용하여 모델을 훈련시키고 tarin 의 20% 를 vaildation set 으로 지정하여 validation을 진행하게 코드를 셋팅 하였다. 이후 test 데이터 데이터에 대해 prediction하여 acurracy를 확인해 보았다.

In [52]:
```
def build_model():
    sequences = layers.Input(shape=(maxlen,))
    embedded = layers.Embedding(max_features, 64)(sequences)
    x = layers.Conv1D(64, 3, activation='relu')(embedded)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPool1D(3)(x)
    x = layers.Conv1D(64, 5, activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPool1D(5)(x)
    x = layers.Conv1D(64, 5, activation='relu')(x)
    x = layers.GlobalMaxPool1D()(x)
    x = layers.Flatten()(x)
    x = layers.Dense(100, activation='relu')(x)
    predictions = layers.Dense(1, activation='sigmoid')(x)
    model = models.Model(inputs=sequences, outputs=predictions)
    model.compile(
        optimizer='rmsprop',
        loss='binary_crossentropy',
        metrics=['binary_accuracy']
    )
    return model

model = build_model()
model.summary()
```

```
Model: "model_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_8 (InputLayer)        [(None, 128)]             0

 embedding_7 (Embedding)     (None, 128, 64)           524288

 conv1d_6 (Conv1D)           (None, 126, 64)           12352

 batch_normalization_4 (Batc  (None, 126, 64)          256
 hNormalization)

 max_pooling1d_4 (MaxPooling  (None, 42, 64)           0
 1D)

 conv1d_7 (Conv1D)           (None, 38, 64)            20544

 batch_normalization_5 (Batc  (None, 38, 64)           256
 hNormalization)

 max_pooling1d_5 (MaxPooling  (None, 7, 64)            0
 1D)

 conv1d_8 (Conv1D)           (None, 3, 64)             20544

 global_max_pooling1d_2 (Glo  (None, 64)               0
 balMaxPooling1D)

 flatten_2 (Flatten)         (None, 64)                0

 dense_14 (Dense)            (None, 100)               6500

 dense_15 (Dense)            (None, 1)                 101

=================================================================
Total params: 584,841
Trainable params: 584,585
Non-trainable params: 256
_________________________________________________________________
```

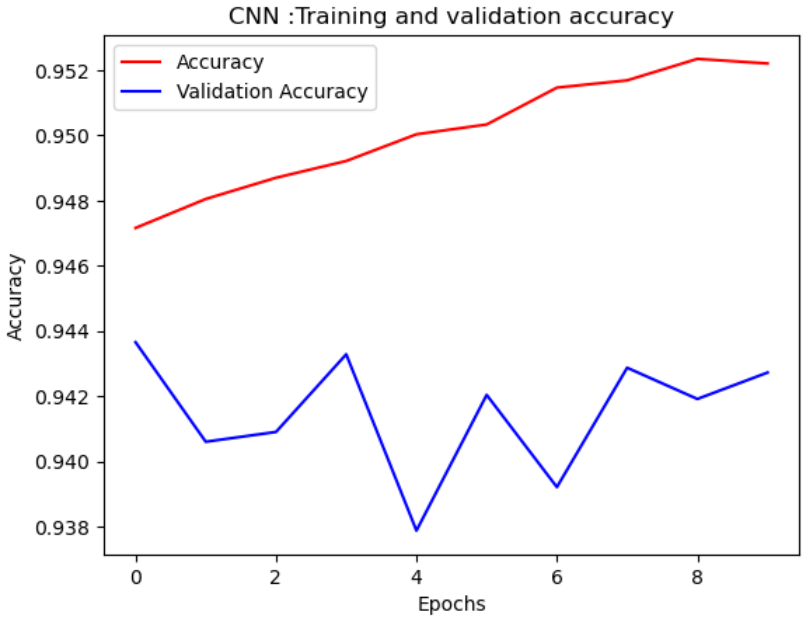
In [72]:
```
history_cnn=model.fit( x_train, y_train, batch_size=64, epochs=10, validation_split=0.20,)
```

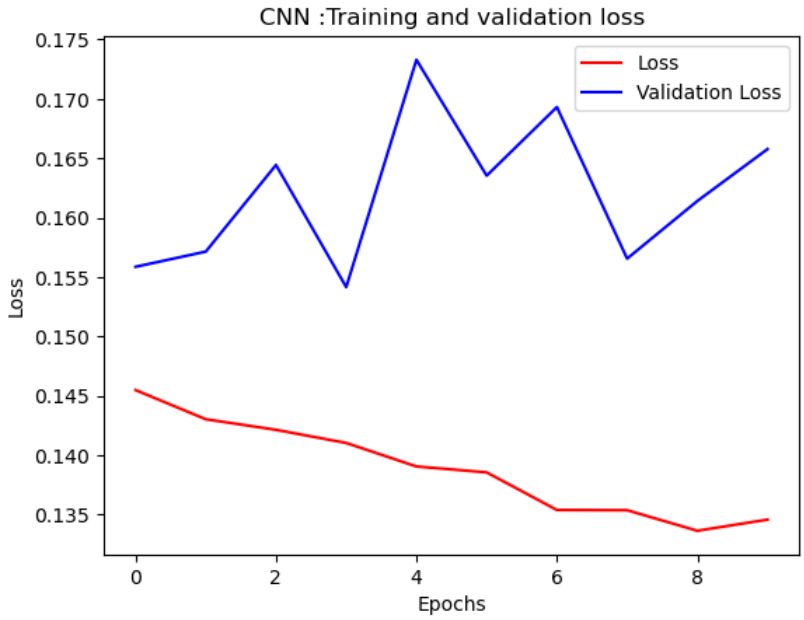
```
Epoch 1/10
45000/45000 [==============================] - 721s 16ms/step - loss: 0.1455 - binary_accuracy: 0.9472 - val_loss: 0.1559 - val_
binary_accuracy: 0.9437
Epoch 2/10
45000/45000 [==============================] - 698s 16ms/step - loss: 0.1430 - binary_accuracy: 0.9480 - val_loss: 0.1571 - val_
binary_accuracy: 0.9406
Epoch 3/10
45000/45000 [==============================] - 688s 15ms/step - loss: 0.1421 - binary_accuracy: 0.9487 - val_loss: 0.1644 - val_
binary_accuracy: 0.9409
Epoch 4/10
45000/45000 [==============================] - 690s 15ms/step - loss: 0.1410 - binary_accuracy: 0.9492 - val_loss: 0.1542 - val_
binary_accuracy: 0.9433
Epoch 5/10
45000/45000 [==============================] - 688s 15ms/step - loss: 0.1391 - binary_accuracy: 0.9500 - val_loss: 0.1733 - val_
binary_accuracy: 0.9379
Epoch 6/10
45000/45000 [==============================] - 694s 15ms/step - loss: 0.1386 - binary_accuracy: 0.9503 - val_loss: 0.1635 - val_
binary_accuracy: 0.9420
Epoch 7/10
45000/45000 [==============================] - 688s 15ms/step - loss: 0.1354 - binary_accuracy: 0.9515 - val_loss: 0.1693 - val_
binary_accuracy: 0.9392
Epoch 8/10
45000/45000 [==============================] - 691s 15ms/step - loss: 0.1354 - binary_accuracy: 0.9517 - val_loss: 0.1566 - val_
binary_accuracy: 0.9429
Epoch 9/10
45000/45000 [==============================] - 691s 15ms/step - loss: 0.1336 - binary_accuracy: 0.9523 - val_loss: 0.1614 - val_
binary_accuracy: 0.9419
Epoch 10/10
45000/45000 [==============================] - 694s 15ms/step - loss: 0.1346 - binary_accuracy: 0.9522 - val_loss: 0.1658 - val_
binary_accuracy: 0.9427
```

In [73]:
```
import matplotlib.image  as mpimg

#-----------------------------------------------------------
# Retrieve a list of list results on training and test data
# sets for each training epoch
#-----------------------------------------------------------
acc=history_cnn.history['binary_accuracy']
val_acc=history_cnn.history['val_binary_accuracy']
loss=history_cnn.history['loss']
val_loss=history_cnn.history['val_loss']

epochs=range(len(acc)) # Get number of epochs

#------------------------------------------------
# Plot training and validation accuracy per epoch
#------------------------------------------------
plt.plot(epochs, acc, 'r')
plt.plot(epochs, val_acc, 'b')
```

```
plt.title('CNN :Training and validation accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["Accuracy", "Validation Accuracy"])

plt.figure()

#--------------------------------------------
# Plot training and validation loss per epoch
#--------------------------------------------
plt.plot(epochs, loss, 'r')
plt.plot(epochs, val_loss, 'b')
plt.title('CNN :Training and validation loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(["Loss", "Validation Loss"])

plt.figure()
```

Out[73]:
```
<Figure size 640x480 with 0 Axes>
```




```
<Figure size 640x480 with 0 Axes>
```

> 10 epoch 동안의 훈련 데이터의 손실은 감소하고 검증 데이터의 손실은 감소하기 보다는 진동한는 경향을 보이고 있다. 또한 정확도 부분에서도 훈련 데이터의 정확도는 epoch가 증가 함에 따라 증가하지만, 손실데이터는 진동한다. 이럴 경우 epoch가 작아서 정확히 수렴하지 않을 수도 있고, 또한 cnn 구조가 이 데이터에 적합하지 않다고 판단 할 수 도 있다.

In [74]:
```
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score

cnn_preds = model.predict(x_test)

accuracy_cnn = accuracy_score(y_test, 1 * (cnn_preds > 0.5))
f1_cnn = f1_score(y_test, 1 * (cnn_preds > 0.5))
```

```
rocauc_cnn = roc_auc_score(y_test, cnn_preds)

print('Accuracy score of the CNN Model: {:0.3}'.format(accuracy_cnn))
print('F1 score of the CNN Model: {:0.3}'.format(f1_cnn))
print('ROC AUC score of the CNN Model: {:0.3}'.format(rocauc_cnn))
```

```
12500/12500 [==============================] - 49s 4ms/step
Accuracy score of the CNN Model: 0.941
F1 score of the CNN Model: 0.941
ROC AUC score of the CNN Model: 0.984
```

> 하지만 위의 플랏과 다르게 test데이터에 대한 예측은 아주 잘되고 있다.

In [ ]:

## 2.2 RNN

RNN은 "Recurrent Neural Network"의 약자로, 순환 신경망이라고도 불린다. 이는 주로 시퀀스 데이터를 처리하는 데 사용되는 신경망 아키텍로, 이전 단계에서 계산된 결과를 현재 단계의 입력으로 반복적으로 전달하여 순환적인 구조를 형성한다.

RNN은 시퀀스 데이터의 특성을 이해하기 위해 시간적인 정보를 고려하여 각 단계에서 RNN은 현재 입력과 이전 단계의 은닉 상태를 함께 출력을 계산한다. 이전 단계의 정보가 현재 단계로 전달되기 때문에 RNN은 시퀀스의 장기 의존성을 학습할 수 있고 이러한 특성은 자연어 처리와 음성 인식과 같은 작업에 유용하게 된다.

RNN은 기본적으로 동일한 가중치를 모든 시간 단계에서 재사용하며, 재귀적인 구조를 가지고 있기 때문에 오래된 정보를 유지하면서 새로운 입력에 대한 정보를 반영할 수 있다. 그러나 RNN에는 *장기 의존성 문제* 가 있어, 긴 시퀀스에서는 이전 정보를 제대로 기억하지 못하는 경우가 발생하기도 한다.

이러한 문제를 해결하기 위해 LSTM(Long Short-Term Memory)과 GRU(Gated Recurrent Unit)와 같은 RNN의 변형 모델이 개발되었고. LSTM과 GRU는 RNN의 기본 구조를 확장하여 장기 의존성을 더 잘 학습할 수 있도록 도와준다

---

- **사용한 RNN 모델**

Embedding 레이어, 2개의 simpleRNN 레이어, 2개의 Dense 레이어로 구성된 RNN 모델로 사용했다.

이 모델은 분류 문제이므로 옵티마이저로는 RMSPROP을 사용하고, 손실 함수로는 이진 크로스엔트로피(Binary Crossentropy)를 사용했다.

train 데이터를 이용하여 모델을 훈련시키고 tarin 의 20% 를 vaildation set 으로 지정하여 validation을 진행하게 코드를 셋팅 하였다. 이후 test 데이터 데이터에 대해 prediction하여 acurracy를 확인해 보았다.

In [32]:
```
def build_rnn_model():
    sequences = layers.Input(shape=(maxlen,))
    embedded = layers.Embedding(max_features, 64)(sequences)
    x = layers.SimpleRNN(128, return_sequences=True)(embedded)
    x = layers.SimpleRNN(128)(x)
    x = layers.Dense(32, activation='relu')(x)
    x = layers.Dense(100, activation='relu')(x)
    predictions = layers.Dense(1, activation='sigmoid')(x)
    model2 = models.Model(inputs=sequences, outputs=predictions)
    model2.compile(
        optimizer='rmsprop',
        loss='binary_crossentropy',
        metrics=['binary_accuracy']
    )
    return model2

rnn_model = build_rnn_model()
rnn_model.summary()
```

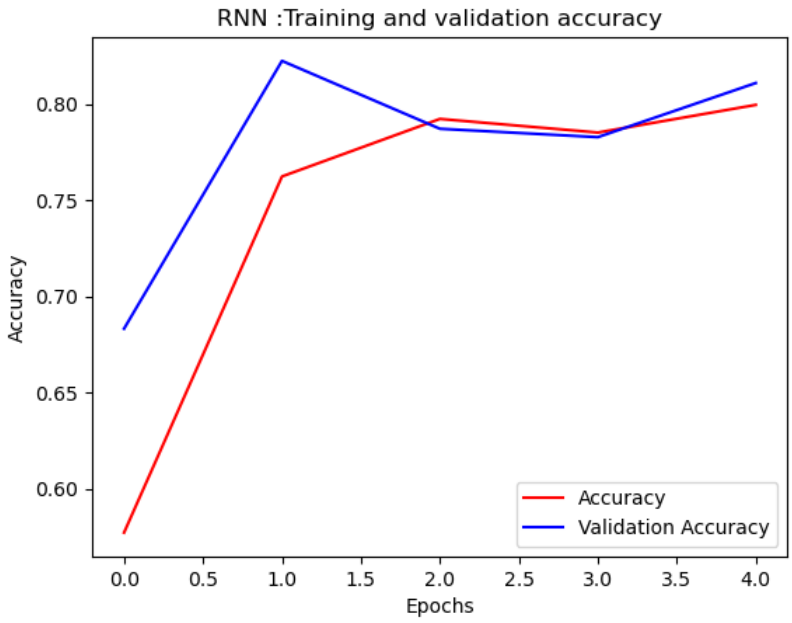
```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 128)]             0

 embedding_2 (Embedding)     (None, 128, 64)           524288

 simple_rnn (SimpleRNN)      (None, 128, 128)          24704

 simple_rnn_1 (SimpleRNN)    (None, 128)               32896

 dense_4 (Dense)             (None, 32)                4128

 dense_5 (Dense)             (None, 100)               3300

 dense_6 (Dense)             (None, 1)                 101

=================================================================
Total params: 589,417
Trainable params: 589,417
Non-trainable params: 0
_________________________________________________________________
```

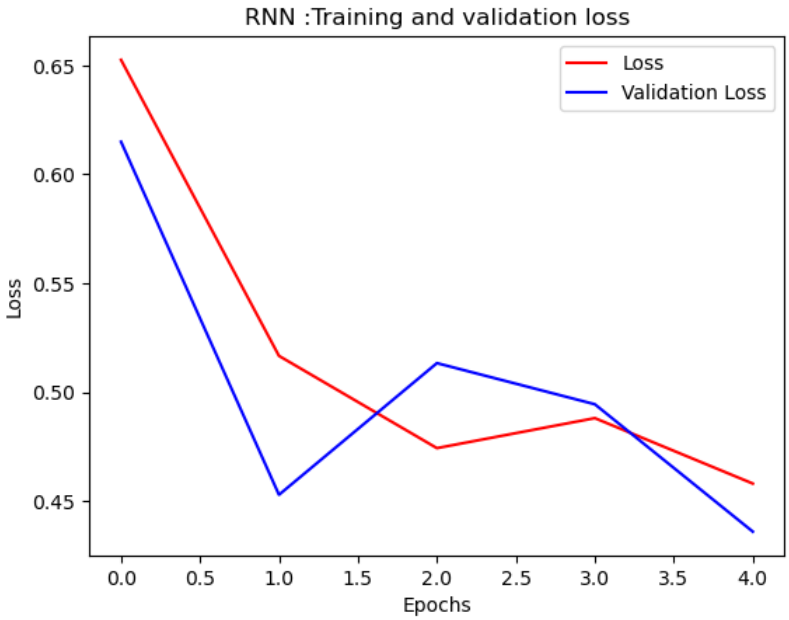
In [33]:
```
rnn_history=rnn_model.fit(x_train, y_train,  batch_size=128, epochs=5, validation_split=0.20, )
```

```
Epoch 1/5
28125/28125 [==============================] - 2597s 92ms/step - loss: 0.6525 - binary_accuracy: 0.5772 - val_loss: 0.6149 - val
_binary_accuracy: 0.6832
Epoch 2/5
28125/28125 [==============================] - 2509s 89ms/step - loss: 0.5166 - binary_accuracy: 0.7624 - val_loss: 0.4529 - val
_binary_accuracy: 0.8225
Epoch 3/5
28125/28125 [==============================] - 2466s 88ms/step - loss: 0.4743 - binary_accuracy: 0.7923 - val_loss: 0.5133 - val
_binary_accuracy: 0.7872
Epoch 4/5
28125/28125 [==============================] - 2459s 87ms/step - loss: 0.4880 - binary_accuracy: 0.7852 - val_loss: 0.4944 - val
_binary_accuracy: 0.7828
Epoch 5/5
28125/28125 [==============================] - 2421s 86ms/step - loss: 0.4580 - binary_accuracy: 0.7996 - val_loss: 0.4359 - val
_binary_accuracy: 0.8110
```

In [58]:
```
import matplotlib.image  as mpimg

#-----------------------------------------------------------
# Retrieve a list of list results on training and test data
# sets for each training epoch
#-----------------------------------------------------------
acc=rnn_history.history['binary_accuracy']
val_acc=rnn_history.history['val_binary_accuracy']
loss=rnn_history.history['loss']
val_loss=rnn_history.history['val_loss']

epochs=range(len(acc)) # Get number of epochs

#------------------------------------------------
# Plot training and validation accuracy per epoch
#------------------------------------------------
plt.plot(epochs, acc, 'r')
plt.plot(epochs, val_acc, 'b')
plt.title('RNN :Training and validation accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["Accuracy", "Validation Accuracy"])

plt.figure()

#------------------------------------------------
# Plot training and validation loss per epoch
#------------------------------------------------
plt.plot(epochs, loss, 'r')
plt.plot(epochs, val_loss, 'b')
plt.title('RNN :Training and validation loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(["Loss", "Validation Loss"])

plt.figure()
```

Out[58]:
```
<Figure size 640x480 with 0 Axes>
```






<Figure size 640x480 with 0 Axes>

> 5 epoch 동안의 훈련 검증 데이터 셋의 손실은 훈련 검증 모두 감소하는 것으로 보이며 정확도 또한 epoch가 커질수록 증가하고 있다. 단 시간상의 문제로 5 epoch 밖에 돌려 보지 못했기에 좀 더 많은 횟수로 돌렸을 때 어떻게 수렴 하는지 볼 필요성이 있다.

In [34]:
```
rnn_preds = rnn_model.predict(x_test)

accuracy_rnn = accuracy_score(y_test, 1 * (rnn_preds > 0.5))
f1_rnn = f1_score(y_test, 1 * (rnn_preds > 0.5))
rocauc_rnn = roc_auc_score(y_test, rnn_preds)

print('Accuracy score of the RNN Model: {:0.3}'.format(accuracy_rnn))
print('F1 score of the RNN Model: {:0.3}'.format(f1_rnn))
print('ROC AUC score of the RNN Model: {:0.3}'.format(rocauc_rnn))
```

```
12500/12500 [==============================] - 182s 15ms/step
Accuracy score of the RNN Model: 0.811
F1 score of the RNN Model: 0.813
ROC AUC score of the RNN Model: 0.872
```

> test 데이터에 대한 예측 정확도는 이전 cnn모델 보다 좋지는 않다. rnn 모델이 cnn 모델보다 이 데이터셋에 안 맞는것 처럼보인다.

In [ ]:
```
```

## 2.3 LSTM

LSTM은 "Long Short-Term Memory"의 약자로, RNN의 변형 중 하나이다. LSTM은 RNN의 단점인 장기 의존성 문제를 해결하고자 고안되었다. 장기 의존성 문제란, RNN이 긴 시퀀스에서 이전 정보를 적절하게 기억하지 못하는 문제를 의미한다.

LSTM은 이 문제를 해결하기 위해 게이트 매커니즘을 도입했는데, 입력 게이트, 삭제 게이트, 출력 게이트 등의 게이트를 사용하여 이전 정보를 얼마나 기억하고 얼마나 새로운 정보를 반영할지를 조절하게 된다. 이를 통해 LSTM은 긴 시퀀스에서도 장기적인 의존성을 학습할 수 있다.

LSTM은 감성 분석과 같은 NLP 작업에 유용하게 사용되는데 이는 LSTM이 시퀀스 데이터를 처리하고 문맥 정보를 유지하면서 텍스트의 감정 표현을 이해하는 데 도움이 되기 때문이다.

---

- **사용한 RNN 모델**

Embedding 레이어, 2개의 LSTM 레이어, 2개의 Dense 레이어로 구성된 LSTM 모델로 사용했다. 마지막 dense layer는 sigmoid를 activation 함수로 사용하였다.이는 이진 분류 작업에 대한 예측된 클래스 확률을 나타내는 하나의 출력 값을 생성하게 한다.

이 모델은 분류 문제이므로 옵티마이저로는 RMSPROP을 사용하고, 손실 함수로는 이진 크로스엔트로피(Binary Crossentropy)를 사용했다.

또한 early stopping 콜백을 이용하여 3 epoch 동안 모니터링 지표가 개선 되지 않으면 훈련을 중지하게 설정하였다.

train 데이터를 이용하여 모델을 훈련시키고 tarin 의 20% 를 vaildation set 으로 지정하여 validation을 진행하게 코드를 셋팅 하였다. 이후 test 데이터 데이터에 대해 prediction하여 acurracy를 확인해 보았다.

In [42]:
```
from tensorflow.keras.callbacks import EarlyStopping

def build_LSTM_model():
    sequences = layers.Input(shape=(maxlen,))
    embedded = layers.Embedding(max_features, 32)(sequences)
```

```
    x = layers.LSTM(32, return_sequences=True)(embedded)
    x = layers.LSTM(32)(x)
    predictions = layers.Dense(1, activation='sigmoid')(x)
    model2 = models.Model(inputs=sequences, outputs=predictions)
    model2.compile(
        optimizer='rmsprop',
        loss='binary_crossentropy',
        metrics=['binary_accuracy']
    )
    return model2

lstm_model = build_LSTM_model()
lstm_model.summary()
```

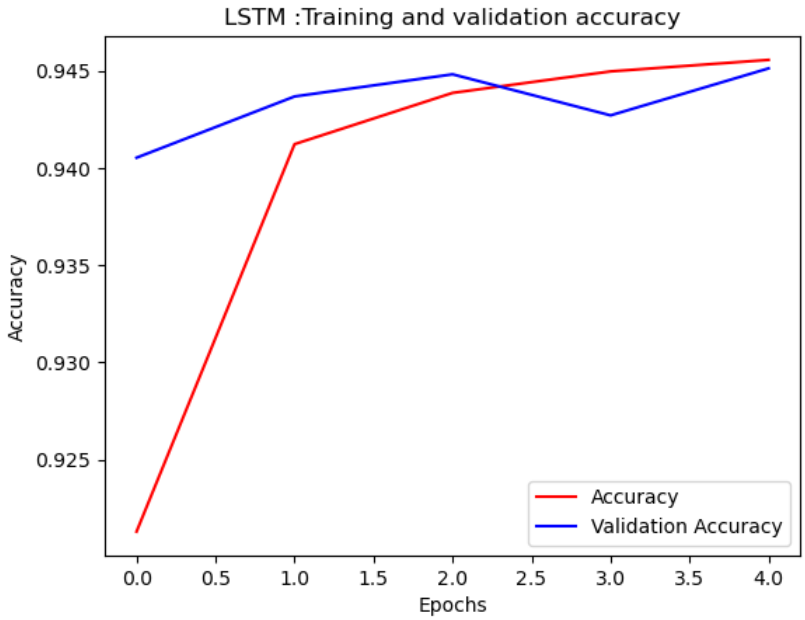
```
Model: "model_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 128)]             0

 embedding_6 (Embedding)     (None, 128, 32)           262144

 lstm_6 (LSTM)               (None, 128, 32)           8320

 lstm_7 (LSTM)               (None, 32)                8320

 dense_13 (Dense)            (None, 1)                 33

=================================================================
Total params: 278,817
Trainable params: 278,817
Non-trainable params: 0
_________________________________________________________________
```

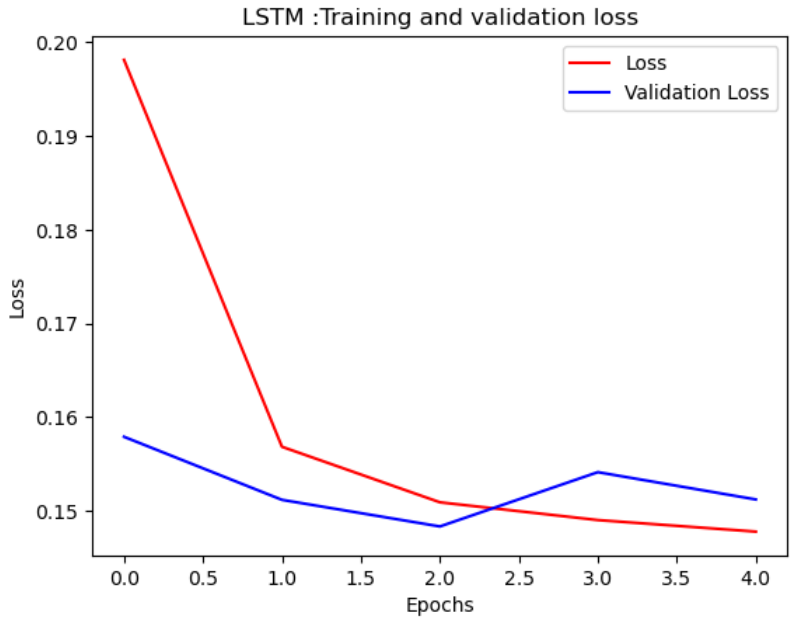
In [43]:
```
history_lstm=lstm_model.fit(x_train, y_train,  batch_size=32, epochs=5, validation_split=0.20,
                    callbacks=[EarlyStopping(monitor='val_loss', patience=3, min_delta=0.0001)])
```

```
Epoch 1/5
90000/90000 [==============================] - 2960s 33ms/step - loss: 0.1981 - binary_accuracy: 0.9213 - val_loss: 0.1579 - val
_binary_accuracy: 0.9405
Epoch 2/5
90000/90000 [==============================] - 2965s 33ms/step - loss: 0.1568 - binary_accuracy: 0.9412 - val_loss: 0.1512 - val
_binary_accuracy: 0.9437
Epoch 3/5
90000/90000 [==============================] - 2971s 33ms/step - loss: 0.1509 - binary_accuracy: 0.9439 - val_loss: 0.1483 - val
_binary_accuracy: 0.9448
Epoch 4/5
90000/90000 [==============================] - 2976s 33ms/step - loss: 0.1490 - binary_accuracy: 0.9450 - val_loss: 0.1541 - val
_binary_accuracy: 0.9427
Epoch 5/5
90000/90000 [==============================] - 2961s 33ms/step - loss: 0.1478 - binary_accuracy: 0.9456 - val_loss: 0.1512 - val
_binary_accuracy: 0.9451
```

In [59]:
```
import matplotlib.image  as mpimg

#-----------------------------------------------------------
# Retrieve a list of list results on training and test data
# sets for each training epoch
#-----------------------------------------------------------
acc=history_lstm.history['binary_accuracy']
val_acc=history_lstm.history['val_binary_accuracy']
loss=history_lstm.history['loss']
val_loss=history_lstm.history['val_loss']

epochs=range(len(acc)) # Get number of epochs

#------------------------------------------------
# Plot training and validation accuracy per epoch
#------------------------------------------------
plt.plot(epochs, acc, 'r')
plt.plot(epochs, val_acc, 'b')
plt.title('LSTM :Training and validation accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["Accuracy", "Validation Accuracy"])

plt.figure()

#------------------------------------------------
# Plot training and validation loss per epoch
#------------------------------------------------
plt.plot(epochs, loss, 'r')
plt.plot(epochs, val_loss, 'b')
plt.title('LSTM :Training and validation loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(["Loss", "Validation Loss"])

plt.figure()
```

Out[59]:
```
<Figure size 640x480 with 0 Axes>
```




```
<Figure size 640x480 with 0 Axes>
```

> LSTM 또한 5 epoch 동안의 훈련 검증 데이터 셋의 손실은 훈련 검증 모두 감소하는 것으로 보이며 정확도 또한 epoch가 커질수록 증가하고 있다. 단 시간상의 문제로 5 epoch 밖에 돌려 보지 못했기에 좀 더 많은 횟수로 돌렸을 때 어떻게 수렴 하는지 볼 필요성이 있다.

In [75]:
```
lstm_preds = lstm_model.predict(x_test)

accuracy_lstm = accuracy_score(y_test, 1 * (lstm_preds  > 0.5))
f1_lstm = f1_score(y_test, 1 * (lstm_preds > 0.5))
rocauc_lstm = roc_auc_score(y_test, lstm_preds)

print('Accuracy score of the LSTM Model: {:0.3}'.format(accuracy_lstm))
print('F1 score of the LSTM Model: {:0.3}'.format(f1_lstm))
print('ROC AUC score of the LSTM Model: {:0.3}'.format(rocauc_lstm))
```

```
12500/12500 [==============================] - 210s 17ms/step
Accuracy score of the LSTM Model: 0.944
F1 score of the LSTM Model: 0.943
ROC AUC score of the LSTM Model: 0.985
```

> LSTM 모델의 경우 RNN모델과 CNN모델 보다 예측값의 정확도가 좋지만, CNN모델과는 크게 차이 나지 않는다.

In [ ]:
```
```

## 2.4 CNN+LSTM

CNN과 LSTM을 결합한 모델은 CNN-LSTM 모델이라고 불리며, 이미지나 텍스트와 같은 시퀀스 데이터에 대해 효과적인 모델이다.

이 모델은 CNN 레이어를 통해 특징 추출을 수행하고, 추출된 특징을 LSTM 레이어로 전달하여 시퀀스적인 특징을 학습하여 시간적인 의존성과 지역적인 구조 모두를 이해할 수 있다. CNN-LSTM 모델은 자연어 처리, 영상 분류, 행동 인식 등 다양한 작업에 활용되고 있다.

---

- **사용한 RNN 모델**

Embedding 레이어, 1개의 CNN 레이어, 2개의 Dense 레이어로 구성된 LSTM 모델로 사용했다. 마지막 dense layer는 sigmoid를 activation 함수로 사용하였다.이는 이진 분류 작업에 대한 예측된 클래스 확률을 나타내는 하나의 출력 값을 생성하게 한다. 신경망의 과대적합을 해결하기 위해 Drop out을 설정하여 활성화 출력은 0.25 1/4로 낮추었다.

이 모델은 분류 문제이므로 옵티마이저로는 RMSPROP을 사용하고, 손실 함수로는 이진 크로스엔트로피(Binary Crossentropy)를 사용했다.

train 데이터를 이용하여 모델을 훈련시키고 tarin 의 20% 를 vaildation set 으로 지정하여 validation을 진행하게 코드를 셋팅 하였다. 이후 test 데이터 데이터에 대해 prediction하여 acurracy를 확인해 보았다.

In [61]:
```
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.layers import Dense, Embedding, Input, Conv1D, GlobalMaxPool1D, Dropout,concatenate, Layer, InputSpec

def build_cnnlstm_model():
    sequences = layers.Input(shape=(maxlen,))
    embedded = layers.Embedding(max_features, 64)(sequences)
    x = Dropout(0.25)(embedded)
    x = Conv1D(32, 3, activation='relu')(x)
    x = layers.LSTM(32, return_sequences=True)(x)
    x = layers.LSTM(32)(x)
    predictions = layers.Dense(1, activation='sigmoid')(x)
    model2 = models.Model(inputs=sequences, outputs=predictions)
    model2.compile(
        optimizer='rmsprop',
        loss='binary_crossentropy',
        metrics=['binary_accuracy']
    )
    return model2


cnnlstm_model = build_cnnlstm_model()
cnnlstm_model.summary()
```

```
Model: "model_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_9 (InputLayer)        [(None, 128)]             0

 embedding_8 (Embedding)     (None, 128, 64)           524288

 dropout (Dropout)           (None, 128, 64)           0

 conv1d_9 (Conv1D)           (None, 126, 32)           6176

 lstm_8 (LSTM)               (None, 126, 32)           8320

 lstm_9 (LSTM)               (None, 32)                8320

 dense_16 (Dense)            (None, 1)                 33

=================================================================
Total params: 547,137
Trainable params: 547,137
Non-trainable params: 0
_________________________________________________________________
```

In [62]:
```
weight_path="early_weights.hdf5"
checkpoint = ModelCheckpoint(weight_path, monitor='val_loss', verbose=1, save_best_only=True, mode='min')
early_stopping = EarlyStopping(monitor="val_loss", mode="min", patience=5)
callbacks = [checkpoint, early_stopping]
```

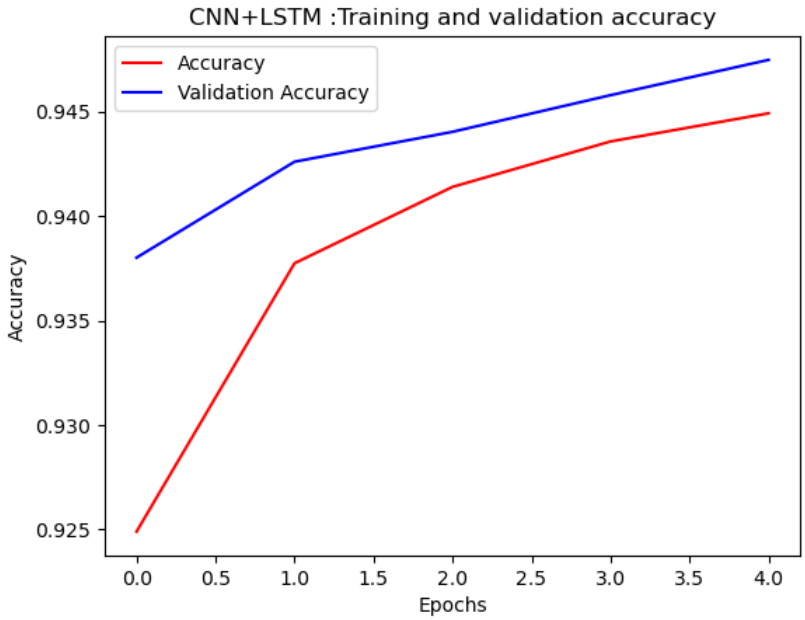
In [67]:
```
history_cnnlstm=cnnlstm_model.fit(x_train, y_train, batch_size=128,
                                  epochs=5, shuffle = True, validation_split=0.20, callbacks=callbacks)
```

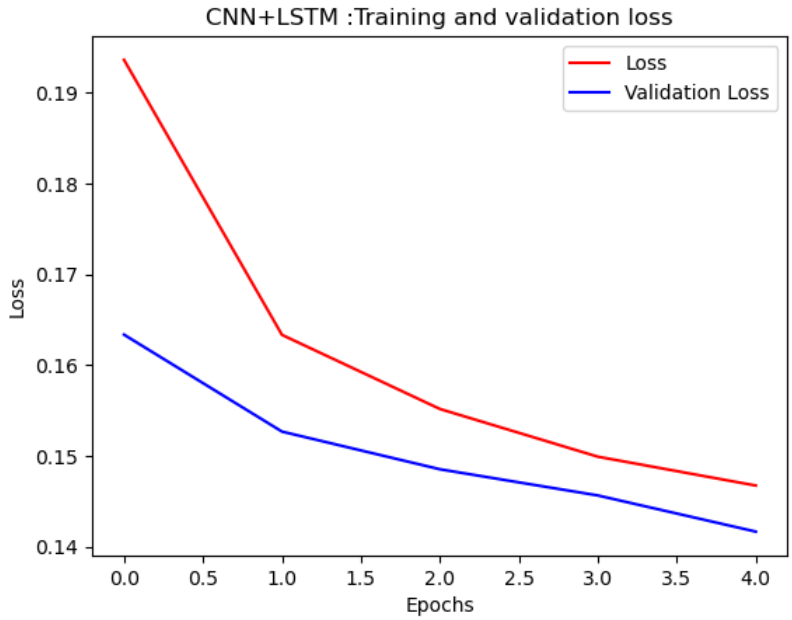
```
Epoch 1/5
22500/22500 [==============================] - ETA: 0s - loss: 0.1936 - binary_accuracy: 0.9249
Epoch 1: val_loss improved from inf to 0.16334, saving model to early_weights.hdf5
22500/22500 [==============================] - 1821s 81ms/step - loss: 0.1936 - binary_accuracy: 0.9249 - val_loss: 0.1633 - val
_binary_accuracy: 0.9380
Epoch 2/5
22500/22500 [==============================] - ETA: 0s - loss: 0.1633 - binary_accuracy: 0.9377
Epoch 2: val_loss improved from 0.16334 to 0.15267, saving model to early_weights.hdf5
22500/22500 [==============================] - 1837s 82ms/step - loss: 0.1633 - binary_accuracy: 0.9377 - val_loss: 0.1527 - val
_binary_accuracy: 0.9426
Epoch 3/5
22500/22500 [==============================] - ETA: 0s - loss: 0.1552 - binary_accuracy: 0.9414
Epoch 3: val_loss improved from 0.15267 to 0.14853, saving model to early_weights.hdf5
22500/22500 [==============================] - 1920s 85ms/step - loss: 0.1552 - binary_accuracy: 0.9414 - val_loss: 0.1485 - val
_binary_accuracy: 0.9440
Epoch 4/5
22500/22500 [==============================] - ETA: 0s - loss: 0.1499 - binary_accuracy: 0.9436
Epoch 4: val_loss improved from 0.14853 to 0.14564, saving model to early_weights.hdf5
22500/22500 [==============================] - 1834s 82ms/step - loss: 0.1499 - binary_accuracy: 0.9436 - val_loss: 0.1456 - val
_binary_accuracy: 0.9458
Epoch 5/5
22500/22500 [==============================] - ETA: 0s - loss: 0.1467 - binary_accuracy: 0.9449
Epoch 5: val_loss improved from 0.14564 to 0.14167, saving model to early_weights.hdf5
22500/22500 [==============================] - 1709s 76ms/step - loss: 0.1467 - binary_accuracy: 0.9449 - val_loss: 0.1417 - val
_binary_accuracy: 0.9475
```

In [ ]:

In [68]:
```
import matplotlib.image  as mpimg

#-----------------------------------------------------------
# Retrieve a list of list results on training and test data
# sets for each training epoch
#-----------------------------------------------------------
acc=history_cnnlstm.history['binary_accuracy']
val_acc=history_cnnlstm.history['val_binary_accuracy']
loss=history_cnnlstm.history['loss']
val_loss=history_cnnlstm.history['val_loss']

epochs=range(len(acc)) # Get number of epochs

#------------------------------------------------
# Plot training and validation accuracy per epoch
#------------------------------------------------
plt.plot(epochs, acc, 'r')
plt.plot(epochs, val_acc, 'b')
plt.title('CNN+LSTM :Training and validation accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["Accuracy", "Validation Accuracy"])

plt.figure()

#------------------------------------------------
# Plot training and validation loss per epoch
#------------------------------------------------
plt.plot(epochs, loss, 'r')
plt.plot(epochs, val_loss, 'b')
plt.title('CNN+LSTM :Training and validation loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(["Loss", "Validation Loss"])

plt.figure()
```

Out[68]:
```
<Figure size 640x480 with 0 Axes>
```




```
<Figure size 640x480 with 0 Axes>
```

> CNN+LSTM 또한 5 epoch 동안의 훈련 & 검증 데이터 셋의 손실은 훈련 & 검증 모두 감소하는 것으로 보이며 정확도 또한 epoch 가 커질수록 증가하고 있다. 하지만 정확도 부분에서 검증 데이터 셋의 loss는 훈련 셋 보다 더 작게 수렴하고 있고 정확도는 검증 데이터 셋이 더 커지고 있음을 알 수있다 . 이부분에서 랜덤으로 validation 데이터가 구성되었으므로, 재구성이 필요한것 처럼 보인다. 또한 5 epoch 동안 일어나는 것이기 때문에 더 증가한 epoch 동안 수렴할수 있음을 유의해야한다.

In [69]:
```
cnnlstm_preds = cnnlstm_model.predict(x_test)

accuracy_rnn = accuracy_score(y_test, 1 * (cnnlstm_preds > 0.5))
f1_rnn = f1_score(y_test, 1 * (cnnlstm_preds > 0.5))
rocauc_rnn = roc_auc_score(y_test, cnnlstm_preds)

print('Accuracy score of the CNN+LSTM Model: {:0.3}'.format(accuracy_rnn))
print('F1 score of the CNN+LSTM Model: {:0.3}'.format(f1_rnn))
print('ROC AUC score of the CNN+LSTM Model: {:0.3}'.format(rocauc_rnn))
```

```
12500/12500 [==============================] - 205s 16ms/step
Accuracy score of the RNN Model: 0.946
F1 score of the RNN Model: 0.946
ROC AUC score of the RNN Model: 0.986
```

> 이전 결과중 가장 좋은 성과를 보였던 CNN과 LSTM을 합친 모형이 가장 좋은 결과를 보였다.

In [ ]:
```
```

# 3. Conclusion & Limitation

- keras 기반의 다양한 모델을 통해 text data를 4가지 방법으로 분석 해 보았다.
- CNN+LSTM > LSTM > CNN > RNN 순으로 아마존 데이터 텍스트문서의 감성분석에서 좋은 정확도를 보였다.
- 합성곱 신경망과 LSTM의 layer를 함께 사용한 모델의 정확도가 더 높게 나오긴 했지만, 시간비용과 모델의 단순함 측면에서 LSTM의 성과가 CNN+LSTM 합친 모형에 비하여 차이가 크지 않기 때문에 모형을 선택 해야한다면 LSTM 모형을 이용 하는 것이 좋을것 같다.
- 프로젝트에 시간이 더 있었다면 많은 횟수의 epoch를 시도 해서 수렴성을 확인하고, hyperparameter를 다양하게 해보아 좀 더 fit하는 값을 해 볼 수 있었겠지만, 시간 관계상 compact하게 프로젝트를 진행한점이 좀 아쉬웠다. 또한 컴퓨터 사양상 좋은 gpu를 기반으로 하지 않았기 때문에 바로 결과 값을 확인 할 수 없었던 한계가 있어 수정이 원활하지 않았다.
- 프로젝트를 진행하면서 hyperparmeter들과 activation함수들에 대한 결과 값차이가 달라 질 수 있기 때문에, 딥러닝에서 hyperparameter 설정은 중요한 요소임을 느끼게 되었다.

---

## + 지각 제출 사유

예심 준비로 인하여 프로젝트 속도가 더뎌지게 되었고, RNN과 LSTM의 수업을 듣고 좀 더 SEQUENTIAL MODEL에대해 알고 난뒤 분석을 시행하고 싶어 온라인 강의 수강 후 제출하게 되었습니다.

또한 TEXT 자료의 양이 방대하다보니 CNN+LSTM 분석시간에도 한 에포크당 30-1시간이 소요 되기도 하여 시간이 더 오래 걸리게 되기도 하였습니다. CNN-LSTM 경우 결과값이 좋게 나왔지만 시간이 오래 걸려 시간이 많이 소요 되었고 그래서 부득이하게 마감기간 보다 더 늦게 제출하게 되었습니다.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: